```
int arr[3] = { };
int arr[2][3] = { {0,1,2}, {3,4,5} };
    arr[2][3] = { 0,1,2,3,4,5 };
```

```
for(size_t i=0; i< vec.size(); i++) { .. }
for(data-type x : vec) { .. }
```

```
vector<int> v1;
vector<int> v1(v2);
vector<int> v1= v2;
vector<int> v1(n);
vector<int> v1(n,val);
vector<int> v1{a,b,c};
vector<int> v1 = {a,b,c};
```

```
for(int i; cin>>i && i !=0) { .. }
```

Declare functions! (header)

— CLASSES

```
# ifndef NAME_H
# define NAME_H
  :
# endif // NAME_H
```

```
Example x;
Example* p = &x;
x.value = ..   ;
(*p).value = ..   ;
p->value = ..   ;
```

getters → CONST

(delegating constructors)

```
Example(int v1, int v2, int v3) : a1(v1), a2(v2), a3(v3) {}
Example(int v1) : Example(v1, 0, 0) {}
Example() : Example(0,0,0) {}
```

operators → take const (references)
CONST (if in-class)

friends:
- in-class declaration of friendship
- out-class declaration (+ definition) of the function

```
Example {
    friend Example operator+(const Example&, const Example&);
    :
};
Example operator+(const Example& lhs, const Example& rhs) { .. }
```

friends → direct access to private members, helper fcts → getters

```
Example {
  public:
    static int function() { .. }
    :
};
int x = Example::function();
Example e;
int y = e.function();
```

— POINTERS/REFERENCES/ITERATORS

```
int x, y;
int* p = &x;
p = &y;
```

Pass-by-reference

1.
```
swap(&a, &b);
void swap(int *p, int *q){
    int temp = *p;
    *p = *q;
}   *q = temp;
```

2.
```
swap(a,b);
void swap(&p, &q){
    int temp = p;
    p = q;
}   q = temp
```

```
vector<int> v1;
auto it1 = v1.begin();
auto it2 = v2.end();          ← one-part-the-last
vector<int>::iterator it3;
vector<int>::const_iterator it4;
for(auto it = v1.begin(); it != v1.end(); it++) {.. *it .. }
```
! `if(v1.begin() != v1.end()) { .. }`

```
for(auto i:v) {..}
for(auto& i : v) {..}
for(const auto& i : v) {..}
```

```
#include<memory>
shared_ptr<int> p = make_shared<int>();   ← default int
shared_ptr<int> p = make_shared<int>(0);  ← init with 0
auto            p = make_shared<int>();
```
! `if(p) {..}` ← "if p is pointing an object"

```
class Dog: public Animal {..}
```
— INHERITANCE & POLYMORPHISM

```
class Base{
  protected:
    int protected_member;
};
class Derived: public Base{
    void function1(Derived&);
    void function2(Base&);
};
void Derived::function1(Derived& d){
    d.protected_member = 2;
}
```
✓
```
void Derived::function2(Base& b){
    b.protected_member = 2;
}
```
✗

(constructors) — inclass definition
```
Base(int v1, int v2): a1(v1), a2(v2){}
  :
Derived(int v1, int v2, int v3, int v4): Base(v1,v2), a3(v3), a4(v4){}
```

(constructors) — .cpp
```
Derived::Derived(int v1, int v2, int v3, int v4): Base::Base(v1,v2), a3(v3), a4(v4){}
```

```
class Base{
  public:
    virtual void function(){..}   /   virtual void function()=0;   ← pure virtual
    virtual ~Base() = default;
};
class Derived: public Base{
  public:
    void function() override;
};
```
! !

in the .cpp: ~~override~~, ~~virtual~~

dynamic binding: 1. virtual, 2. override, 3. pointers/references

```
void final_function(Base& item){.. // call function() that prints the Σ }
Base b(1,2);
Derived d(1,2,3,4);
final_function(b);  → function() ∈ Base    → out: 3
final_function(d);  → function() ∈ Derived → out: 10
```

---

containers of Base & Derived objects:

1. if they already exist: RAW POINTERS

```
Base b(0,0);
Derived d(0,0,0,0);
vector<Base*> container;
container.push_back(&b);
container.push_back(&d);
```

2. if they don't exist yet: SMART PTRS

```
vector<shared_ptr<Base>> container;
container.push_back(make_shared<Base>(0,0));
container.push_back(make_shared<Derived>(0,0,0,0));
```

1./2. we can write: `container[0]->a1 = ..;`

COPY CONTROL

```
class Example{
    public:
        Example() = default;                          ← default constructor
        Example(const Example&) = default;            ← default copy constructor
        Example& operator=(const Example&);
        ~Example() = default;                         ← default destructor
};
Example& Example::operator=(const Example&) = default;   ← default assignment
```

- if ~~default~~ → "delete"
- Remember: Example& will return: `return *this;`

(not default copy constructor)

```
Example::Example(const Example& rhs) : a1(rhs.a1), .. {}
```

CLASSESS & POINTERS

- like-a-value → copies are ⊥, better NO pointers:

```
class Example{
    vector<string> data;
};
```

- like-a-pointer → copies share data, smart ptrs:

```
class Example{
    shared_ptr<vector<string>> data;
};
```

---

implicit type conversions:

```
class Example{
    public:
        explicit Example(int v1): a1(v1){}
};
```

STL

```
vec.reserve(n);      O(vec.size())   → occupies space (empty)
vec.resize(n);       O(n)            → default-fill the new space
vec.push_back(elem); O(vec.size())   } worst, average: O(1)
```

```
container c;
c.back();   } returns a reference to the last/first element
c.front();
c.at(n);    — const
c[n];       — non-const
c.begin();  / c.cbegin();
c.end();    / c.cend();
c.rbegin(); / c.crbegin();
c.rend();   / c.crend();
```

```
swap(c1,c2);
c.size();
c.max_size();
c.empty();
```

```
map<string, int>  word_count;
string word;
while (cin >> word)
    ++word_count[word];
for (const auto& w: word_count)
    cout << w.first << "occurs" << w.second << (w.second > 1)? "times" : "time";
```

MAP

```
word_count["Hello"] = 1;          it creates it, in case
word_count.at("Hello")            it does NOT create it
word_count.insert(make_pair("Hello", 1));
auto it = word_count.begin();
(*it).second = 4;
it->second = 4;
(*it).first = "Hi";               ✗ key is CONST
```

PAIR

```
pair<int, int>   p1;
pair<int, int>   p2(0,0);
pair<int, int>   p3 = {0,0};
p1.first = ..
p2.second = ..
make_pair(0,0)
```

SET

```
set<string> names;
if (names.find("Mario") == names.end()) {..}

vector<int>  vec = {1,1,2,2,3,3};
set<int>     set_vec(vec.begin(), vec.end());
```

MAP/SET

```
c.insert(v)          v key (set), pair (map)
c.insert(b,e)
c.erase(k)           k key
c.erase(b,e)
c.find(k)
c.count(k)
c.lower_bound(k)     iterator to the 1st elem with key >= k
c.upper_bound(k)                                        >
```

```
map<string, int>  ⟶  key_type      = string
                     mapped_type   = int
                     value_type    = pair<const string, int>
```